Common Mistakes When Using the Command Pattern and How to Avoid Them

Apostolos V. Zarras
Department of Computer Science and Engineering, University of Ioannina, Greece
zarras@cs.uoi.gr

ABSTRACT

Command, is a behavioral pattern from the Gang of Four catalog that allows us to structure an application with respect to primitive actions that can be easily managed and executed. The main idea is to decouple the objects that invoke actions from the objects that know how to perform them, by encapsulating everything that is needed for executing the actions in corresponding command objects. The application can comprise different classes of command objects that realize different actions. The different classes of command objects implement the same interface. Therefore, command objects can be passed as parameters to other objects that use them to execute the respective actions, without knowing how this is actually done.

This paper revisits the Command pattern, focusing on the configuration of command objects, when the data that are needed for the execution of the actions become available as soon as the actions should be executed. In this case, it is not clear which class is responsible for configuring the command objects and how this should be done. The paper reports common mistakes when dealing with this problem, as anti-patterns, observed during the project of a software engineering course. The observed mistakes invalidate the benefits of the Command pattern because the invoking objects are explicitly or implicitly coupled with the concrete classes of command objects that realize the different actions. The paper further introduces a pattern that deals with the problem.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**;

KEYWORDS

Behavioral Patterns, Command

ACM Reference Format:
Apostolos V. Zarras. 2020. Common Mistakes When Using the Command Pattern and How to Avoid Them. In *European Conference on Pattern Languages of Programs 2020 (EuroPLoP '20), July 1–4, 2020, Virtual Event, Germany.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3424771.3424793

1 INTRODUCTION

Command is a well-known behavioral pattern, introduced in the Gang of Four (GoF) catalog [6]. At a glance, the pattern decouples an object that *invokes* an action, in the context of a particular application, from a *receiving object* that knows how to perform it.

The idea for decoupling the invoking object from the receiving object is *to encapsulate all the information that is necessary to perform an action* in a *command object*. The required information may comprise *input data* for the action and a reference to the *receiving object* that knows how to perform the action. Then, the command object can be passed as parameter to the invoking object that executes the encapsulated action, without knowing much about it. This way, the invoking object can be easily reused for executing different actions, realized by different concrete classes of command objects that implement the same common interface. Moreover, the application can be easily extended with more actions. The pattern further enables the composition of command object into more complex ones, the logging of command objects that have been performed and the realization of undo/redo operations.

Although the intent of the Command pattern is quite clear, there is a key point that complicates the use of the pattern in many situations, especially when the pattern is applied by inexperienced developers.

> ***Often, the information that is required by a command object is only available at the moment when the invoking object has to execute the respective action. The issue in these cases is which pattern participant will be responsible for the configuration of the command object*.**

This paper discusses frequent mistakes when dealing with the aforementioned problem, as anti-patterns that have been empirically observed during the project of a software engineering course. These anti-patterns, compromise the benefits of the Command pattern, by making the invoking object responsible for the configuration of the command object. Consequently, the invoking object is coupled with the different concrete classes of command objects involved in the application. The paper further introduces a pattern that deals with the configuration of command objects. According to the pattern, the command object configures itself with the required information, whenever there is a need to execute an action, using a reference to an object that is aware of and provides the required information.

The rest of this paper is structured as follows. Section 2, provides some brief background information on patterns and anti-patterns. Section 3, defines the context, the problem of configuring command object dynamically, and the forces involved in this problem. Sections 4 to 6 discuss the observed anti-patterns. Section 7, reports the pattern that provides a better solution to the problem. Section 8,

EuroPLoP '20, July 1–4, 2020, Virtual Event, Germany

ACM ISBN 978-1-4503-7769-0/20/07…$15.00
https://doi.org/10.1145/3424771.3424793

provides details on the pattern mining process and the observed uses of the anti-patterns and the pattern, in the context of the software engineering course project. Finally, Section 9 summarizes the contribution of this paper.

2 BACKGROUND & RELATED WORK

The Gang of Four (GoF) design patterns [6] are a valuable contribution to the software engineering community. Several studies showed that their use is beneficial for improving software quality [2, 7] and maintainability [11]. Explicitly documenting the use of design patterns also facilitates software maintenance [12]. However, the classes that implement design patterns may change during the lifetime of a particular software [3, 4]. Changes may result in new defects. The extent of this problem varies depending on the patterns that the changed classes implement [13]. Another line of research, further showed that the inappropriate use of design patterns may create severe problems [14]. For this reason, several approaches emerged for checking the correct use of design patterns [8].

Anti-patterns report ineffective solutions to frequently encountered problems that introduce risks, inefficiencies and other problems [9]. Anti-patterns may concern several different issues like the software architecture, design, implementation, documentation and so on [1, 5, 10, 15].

In this paper, we focus on issues that result from the misuse of the Command pattern. In particular, we report anti-patterns that concern the configuration of Command objects and a pattern that provides a better solution to this problem.

3 CONTEXT, PROBLEM & FORCES

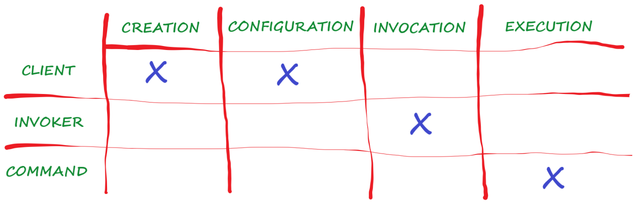
Context

A junior developer implements an object-oriented application. To this end, he structures the application based on the GoF Command pattern.

The general structure of the pattern (Figure 1) that is specified in the GoF catalog [6] involves the following participants:

- Command defines a common interface for different classes of objects that enable the execution of different actions. The interface provides the execute() operation for executing an action.
- The Command interface is implemented by different concrete classes like ConcreteCommandA and ConcreteCommandB that realize respective actions.
- ReceiverA and ReceiverB, provide methods that are used to carry out the actions, realized by ConcreteCommandA and ConcreteCommandB, respectively.
- Client is responsible for *creating* objects of a concrete class that implements the Command interface and for *configuring* these objects with the information that is necessary for performing an action.
- Invoker has a Command reference that can be set to an object of a concrete class (ConcreteCommandA or ConcreteCommandB) that implements the Command interface.

A particular execution scenario that reflects the general structure of the pattern is given below:



	CREATION	CONFIGURATION	INVOCATION	EXECUTION
CLIENT	X	X		
INVOKER			X	
COMMAND				X

Figure 2: Assignment of responsibilities to the pattern participants.

- A Client object creates a ConcreteCommandA object, sets its ReceiverA object and the data that are needed for executing the action.
- The Invoker object stores a reference to the ConcreteCommandA object.
- The Invoker object invokes the execute() method on the referenced object to execute the respective action.
- The ConcreteCommandA object executes the action by invoking methods on the ReceiverA object.

Figure 2 summarizes the assignment of responsibilities to the pattern participants.

Problem

A Command object needs certain data and a receiver object to execute an action. **The required data and receiver object are only available at the moment when the action should be executed.** In this situation, the developer can not implement the pattern exactly as prescribed in the GoF catalog. Specifically, **the Client object can not configure the Command object before giving it to the Invoker object**. Hence, the assignment of responsibilities to participants (Figure 2) must change. The developer should decide **which object is going to be responsible for the configuration of the Command object, instead of the Client object**.

Forces

- For extensibility and reusability reasons, the developer must keep the Invoker class decoupled from the concrete classes that implement the Command interface, and from the receiver classes that realize the actions.
- The developer should further ensure that the Command interface is simple and uniform to facilitate the execution of different kinds of actions.

4 INVOKER-DRIVEN CONFIGURATION

Anti-Pattern

The developer keeps the creation of the Command object at the Client class, but assigns the configuration of the Command object to the Invoker class (Figure 5).

In detail, the Client object creates a Command object that belongs to a particular concrete class (Figure 3). The Invoker object stores a reference to the Command object. To execute an action, the Invoker object determines the concrete class (ConcreteCommandA or ConcreteCommandB) of the referenced Command

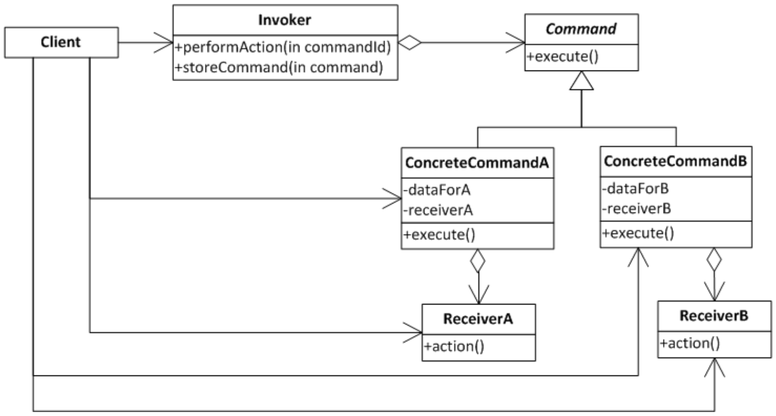
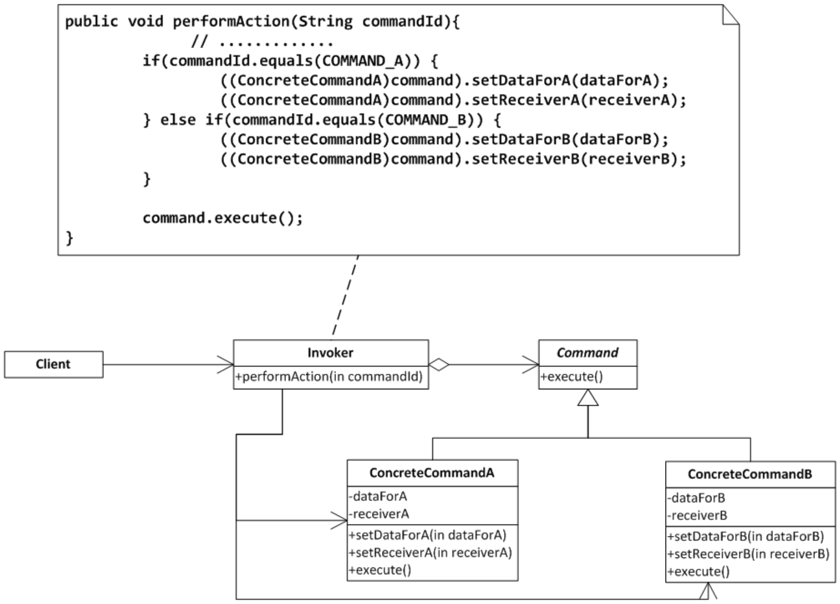


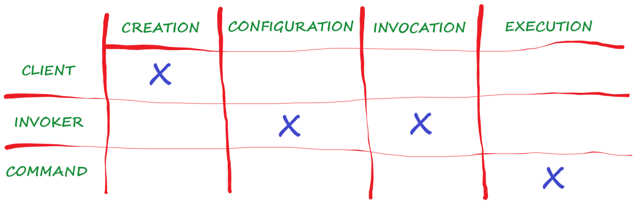
Figure 1: The structure of the Command pattern.



```
public void performAction(String commandId){
              // .............
        if(commandId.equals(COMMAND_A)) {
                ((ConcreteCommandA)command).setDataForA(dataForA);
                ((ConcreteCommandA)command).setReceiverA(receiverA);
        } else if(commandId.equals(COMMAND_B)) {
                ((ConcreteCommandB)command).setDataForB(dataForB);
                ((ConcreteCommandB)command).setReceiverB(receiverB);
        }

        command.execute();
}
```

Client
Invoker
+performAction(in commandId)
Command
+execute()
ConcreteCommandA
-dataForA
-receiverA
+setDataForA(in dataForA)
+setReceiverA(in receiverA)
+execute()
ConcreteCommandB
-dataForB
-receiverB
+setDataForB(in dataForB)
+setReceiverB(in receiverB)
+execute()

Figure 3: Invoker-Driven Configuration.



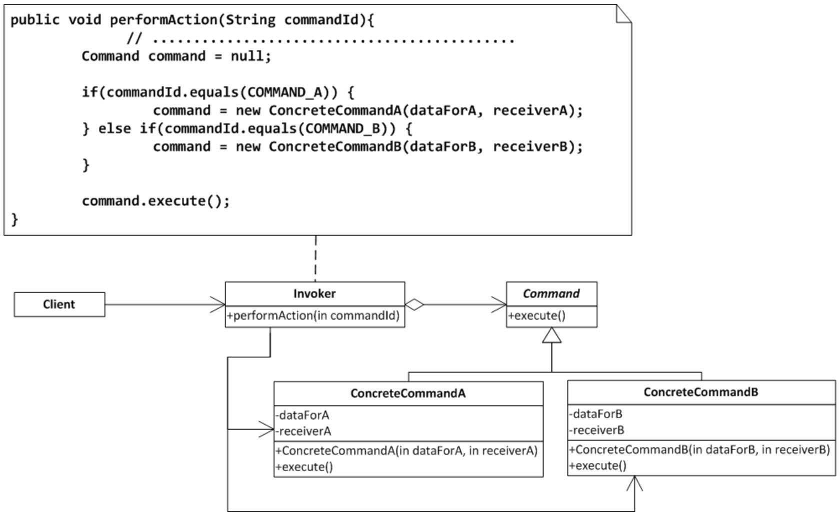
	CREATION	CONFIGURATION	INVOCATION	EXECUTION
CLIENT	X			
INVOKER		X	X	
COMMAND				X

Figure 5: Invoker-Driven Configuration - Responsibilities.

object. Following, it down-casts the Command object reference to a concrete class reference. Using the concrete reference, the Invoker object configures the Command object with the appropriate data and receiver object by calling setter methods, provided by the concrete class. Finally, the Invoker object uses the Command object reference to execute the action, by invoking the execute() method.

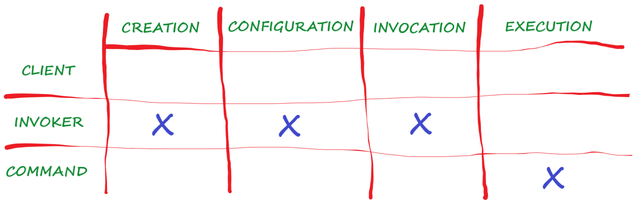
Consequences

- The Invoker class is coupled with the concrete classes (ConcreteCommandA, ConcreteCommandB) that implement the Command interface. In particular, the Invoker object uses the setter methods of these classes to configure the referenced Command object with the required data and receiver object. The Invoker class is further coupled with the receiver classes.



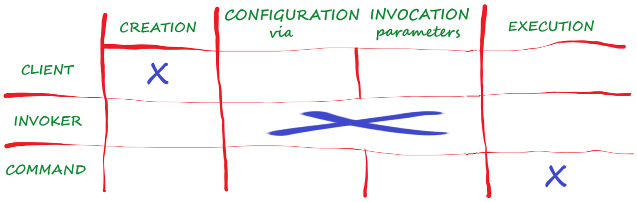
```
public void performAction(String commandId){
        // ...........................................
        Command command = null;

        if(commandId.equals(COMMAND_A)) {
                command = new ConcreteCommandA(dataForA, receiverA);
        } else if(commandId.equals(COMMAND_B)) {
                command = new ConcreteCommandB(dataForB, receiverB);
        }

        command.execute();
}
```

Figure 4: Invoker-Driven Creation and Configuration.



	CREATION	CONFIGURATION	INVOCATION	EXECUTION
CLIENT				
INVOKER	X	X	X	
COMMAND				X

Figure 6: Invoker-Driven Creation and Configuration - Responsibilities.



	CREATION	CONFIGURATION via	INVOCATION parameters	EXECUTION
CLIENT	X			
INVOKER		X		
COMMAND				X

Figure 9: Invoker-Driven Parameterized Invocation - Responsibilities.

- This solution does not introduce any additional complexity to the Command interface.

5 INVOKER-DRIVEN CREATION AND CONFIGURATION

Anti-Pattern

The developer moves the creation and the configuration of the Command object from the Client class to the Invoker class (Figure 6).

Specifically, when there is a need to execute a particular action the Invoker object determines the concrete class (ConcreteCommandA or ConcreteCommandB) that corresponds to the action (Figure 4). Then, the Invoker object creates an object of the concrete class, using a parameterized constructor that is provided by the class. To configure the Command object, the Invoker object passes as parameters to the constructor a receiver object and the data that are needed for the execution of the action. Finally, the Invoker object invokes the execute() method on the newly created Command object to execute the action.

Consequences

- The decoupling of the Invoker class from the concrete classes that implement the Command interface is not possible. In particular, the Invoker class uses the parameterized constructors of the concrete classes that implement the Command interface. Moreover, the Invoker class depends on the classes of the receiver objects.
- The Command interface remains simple and uniform for the different kinds of actions.

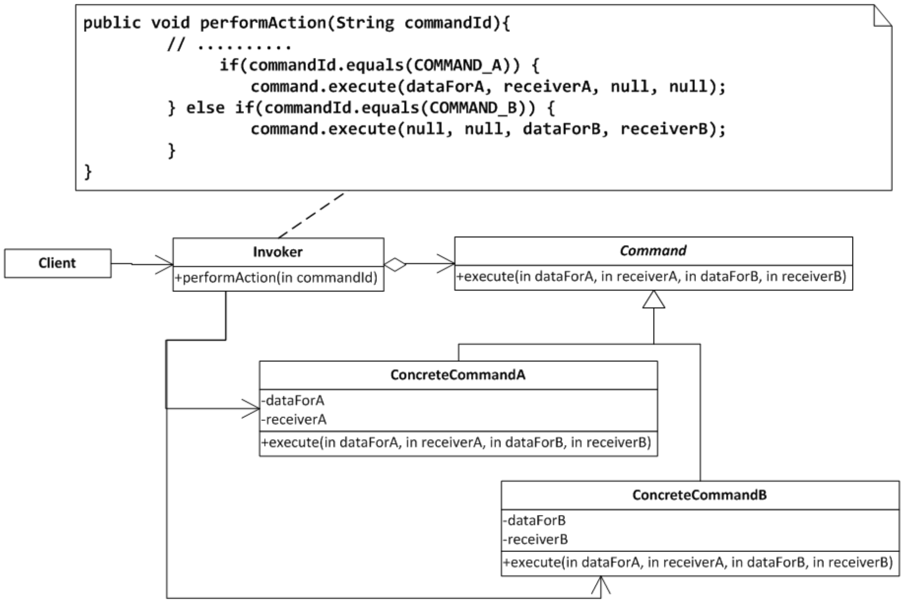
6 INVOKER-DRIVEN PARAMETERIZED INVOCATION

Anti-Pattern

The developer blends the configuration and the invocation of the Command object in a single responsibility that is assigned to the Invoker class (Figure 9).

In detail, the data and the receiver objects, required by the concrete classes that implement the Command interface are passed as parameters to the execute() method. Parameter passing can be implemented in different ways.

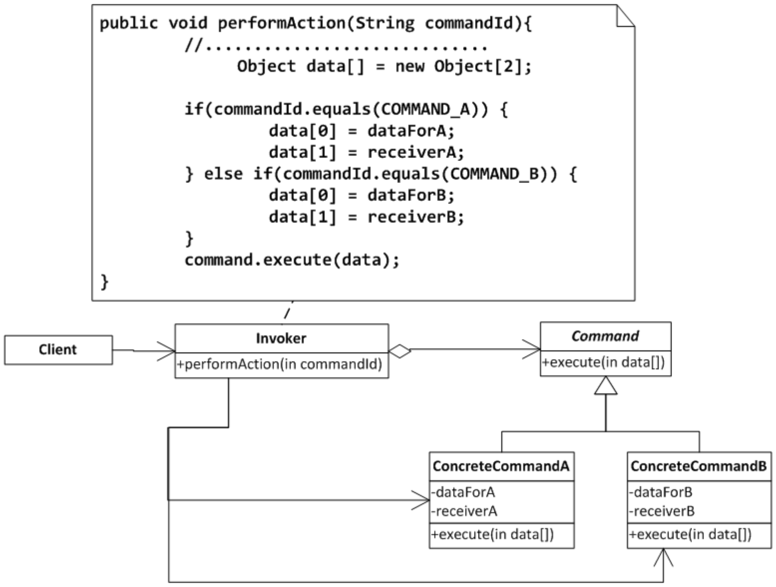
- One possible variant (Figure 7) is to add a parameter list to the execute() method that consists of the union of all the data and receiver objects that are needed by the concrete classes. In this variant, the Client object creates a Command object of a concrete class that implements the Command interface. The Invoker object stores a reference to



```
public void performAction(String commandId){
        // ..........
             if(commandId.equals(COMMAND_A)) {
                 command.execute(dataForA, receiverA, null, null);
        } else if(commandId.equals(COMMAND_B)) {
                 command.execute(null, null, dataForB, receiverB);
        }
}
```

Client
Invoker
+performAction(in commandId)
Command
+execute(in dataForA, in receiverA, in dataForB, in receiverB)
ConcreteCommandA
-dataForA
-receiverA
+execute(in dataForA, in receiverA, in dataForB, in receiverB)
ConcreteCommandB
-dataForB
-receiverB
+execute(in dataForA, in receiverA, in dataForB, in receiverB)

Figure 7: INVOKER-DRIVEN PARAMETERIZED INVOCATION - Long parameter list.

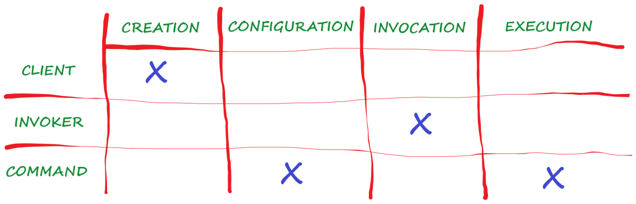


```
public void performAction(String commandId){
        //.............................
              Object data[] = new Object[2];

        if(commandId.equals(COMMAND_A)) {
                data[0] = dataForA;
                data[1] = receiverA;
        } else if(commandId.equals(COMMAND_B)) {
                data[0] = dataForB;
                data[1] = receiverB;
        }
        command.execute(data);
}
```

Figure 8: INVOKER-DRIVEN PARAMETERIZED INVOCATION - Single generic parameter.

the Command object. To execute an action, the Invoker object determines the concrete class (ConcreteCommandA or ConcreteCommandB) of the referenced Command object. Then, the Invoker object invokes the execute() method on the Command object. The Invoker object passes as parameters to the execute() method the data and the receiver object that correspond to the concrete class of the referenced Command object. The remaining parameters are set to null, or to some other default value, depending on the types of the parameters.

- Another possible variant is to add a single parameter to the execute() method (Figure 8). In this variant, the parameter is a generic data structure that can store an arbitrary number of elements of different types. The Client object creates a Command object of a concrete class that implements the Command interface. The Invoker object stores a reference to the Command object. To execute an action, the Invoker



	CREATION	CONFIGURATION	INVOCATION	EXECUTION
CLIENT	X			
INVOKER			X	
COMMAND		X		X

Figure 11: Command Self Configuration - Responsibilities.

object identifies the concrete class (ConcreteCommandA or ConcreteCommandB) of the referenced Command object. Then, the Invoker object stores the data and the receiver object that correspond to the concrete class of the referenced Command object to the generic data structure. Finally, the Invoker object calls the execute() method on the Command object with the generic data structure as parameter.

Consequences

- The Invoker class is not explicitly coupled with the concrete classes (ConcreteCommandA, ConcreteCommandB) that implement the Command interface, in the sense that it does not refer to the concrete classes. However, in both variants of this anti-pattern the Invoker class is implicitly coupled with the concrete classes that implement the Command interface for the following reasons:
  - In the first variant of the anti-pattern, the parameters of the execute() method depend on the data and the receiver object required by each concrete class. The Invoker object provides values for the parameters of the execute() method.
  - In the second variant of the anti-pattern, the Invoker class is implicitly coupled with the concrete classes that implement the Command interface because it prepares the contents of the generic data structure that is given as parameter to execute(), which are specific to the concrete classes.
- The Command interface becomes more complex.
  - In the first variant of the anti-pattern, the parameter list can become very long. The parameter list depends on the different concrete classes that implement the Command interface. Moreover, not all of the parameters are useful for each action.
  - In the second variant of the anti-pattern, the parameter list consists of a single parameter. However, the fact that this parameter is a generic data structure that can store any number of elements of any possible type is an issue concerning type safety and the readability of the code.

7 COMMAND SELF CONFIGURATION

Pattern

The developer moves the configuration of the Command object from the Client class to the concrete classes that implements the Command interface (Figure 11).

The idea is to let a Command object configure itself with the appropriate input data and receiver object at the moment when the action should be executed. To this end, the Command object should have references to certain objects that (1) are available at the moment when the Command object is created and (2) can be used by the Command object to obtain the data and receiver object that are needed, at the moment when the action should be executed.

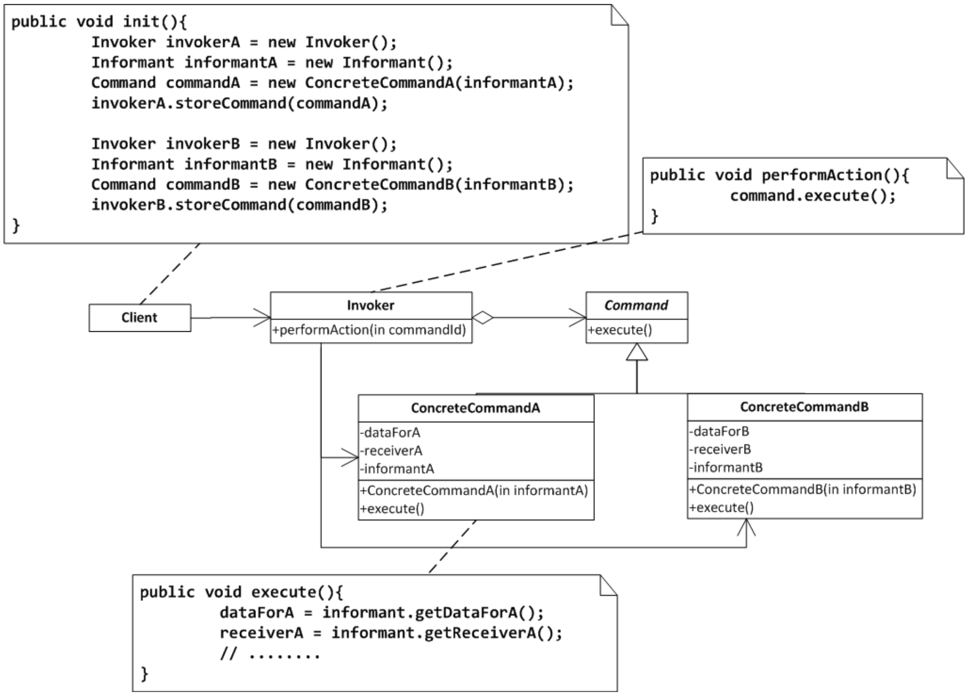
According to this idea, in the solution that is sketched in Figure 10 the Client object creates a Command object and configures it with a reference to an Informant object that is available when the Client object creates the Command object and can provide to the Command object the data and the receiver object that are needed, at the moment when the action should be executed. *Note that in many cases the role of the* Invoker *can be played by the* Client *participant.* When the Invoker object calls the execute() method on the Command object, the latter uses the reference to the Informant object to configure itself with the data and the receiver object that are needed for the execution of the action.

Example

Figure 12 depicts the design of a simple document editor for Latex users. Latex is a well known document preparation markup language. It provides a large variety of styles and commands that enable advanced document formatting. The design of the Latex editor relies on the Command pattern. The editor allows the user to create, save, or load a Latex document. The user can automatically add Latex macros to the document and edit its contents. The editor allows the user to edit multiple documents concurrently.

The GUI provides different menus for the file management and the editing actions of the editor. The provided actions correspond to respective menu items. The menus are objects of the LatexEditorMenu class, while their constituent items are objects of the LatexEditorMenuItem class. Concerning the Command pattern structure, the LatexEditorMenu class plays the role of the Client. On the other hand, the LatexEditorMenuItem class plays the role of the Invoker. Hence, the LatexEditorMenuItem class has a reference to a Command object.

The actions that are provided by the editor are implemented as concrete classes that implement the Command interface. For example, the LoadCommand class implements the action that loads an existing Latex document from disk. The required data for this action is a filename. The loading is done with the help of a DocumentManager object that plays the role of the receiver. Specifically, the DocumentManager object serves for creating a new Document object that holds the contents of the Latex document. The DocumentManager object that plays the role of the receiver is created when the editor starts running and remains the same throughout the execution of the editor. However, the required filename is specified by the user right before the execution of the loading action.

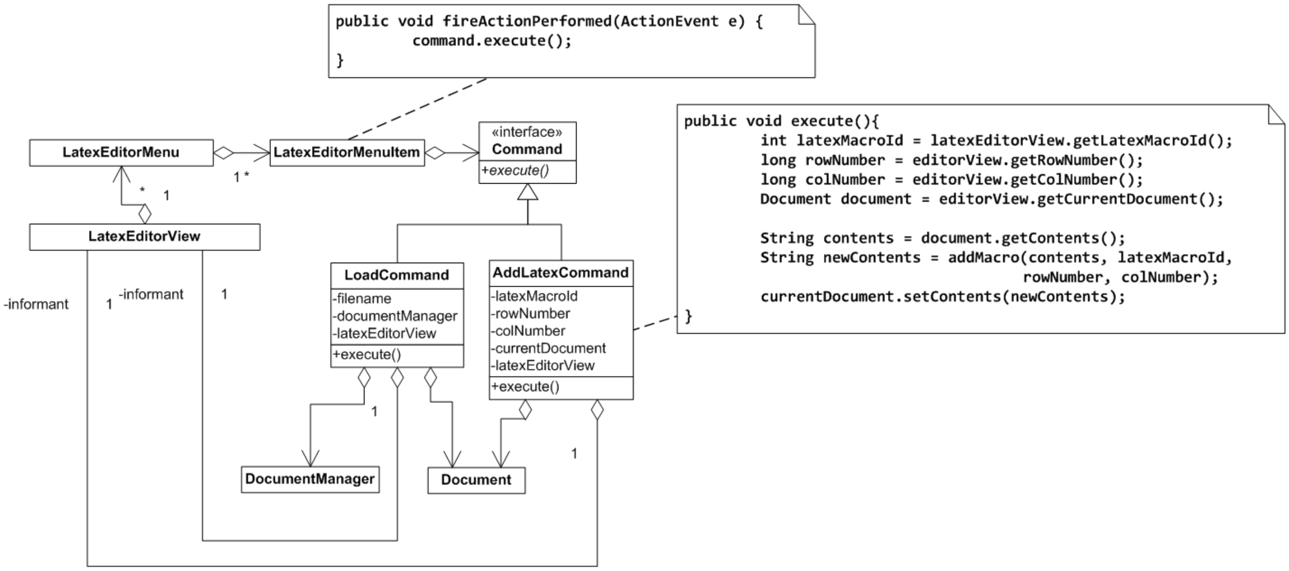


```
public void init(){
        Invoker invokerA = new Invoker();
        Informant informantA = new Informant();
        Command commandA = new ConcreteCommandA(informantA);
        invokerA.storeCommand(commandA);

        Invoker invokerB = new Invoker();
        Informant informantB = new Informant();
        Command commandB = new ConcreteCommandB(informantB);
        invokerB.storeCommand(commandB);
}
```

```
public void performAction(){
        command.execute();
}
```

```
public void execute(){
        dataForA = informant.getDataForA();
        receiverA = informant.getReceiverA();
        // ........
}
```

Figure 10: Command Self Configuration.



```
public void fireActionPerformed(ActionEvent e) {
        command.execute();
}
```

```
public void execute(){
        int latexMacroId = latexEditorView.getLatexMacroId();
        long rowNumber = editorView.getRowNumber();
        long colNumber = editorView.getColNumber();
        Document document = editorView.getCurrentDocument();

        String contents = document.getContents();
        String newContents = addMacro(contents, latexMacroId,
                                      rowNumber, colNumber);
        currentDocument.setContents(newContents);
}
```

Figure 12: Command Self Configuration in the Latex editor example.

The AddLatexCommand implements the action that adds a Latex macro to the contents of the current Latex document that the user is editing. The required data for this action are an identifier of the Latex macro that should be added to the current document, row and column numbers that specify the position where the macro should be added, and a Document object that holds the contents of the current Latex document. Regarding the pattern structure, the Document object plays the role of the receiver for this action. The required data and the receiver object are determined right before the execution of the action.

In the Latex editor project the role of the Informant is played by the LatexEditorView object, which also plays the role of the

Client. Hence, the LatexEditorView object creates a Command object and configures the object with a reference to itself (LatexEditorView object). Then, the LatexEditorView object configures a LatexEditorMenuItem object with a reference to the Command object.

To execute an action, the LatexEditorMenuItem object calls the execute() method on the referenced Command object. The Command object uses the reference to the LatexEditorView object to configure itself with the data and the receiver object that are needed for the execution of the action, at the moment when the action should be executed. For instance, if the Command object belongs to the AddLatexCommand class, it obtains from the LatexEditorView object the Latex macro that should be added to the document, the row and the column numbers that identify the position where the macro should be added, and a Document object that plays the role of the receiver object.

Consequences

- The Invoker class is decoupled from the concrete classes that implement the Command interface.
- The Command interface remains simple and uniform for the different kinds of actions.
- In some cases, the identification of the appropriate Informant object may not be obvious (e.g., it may be an object that knows an object, which refers to another object that can provide the information for the action execution).
- The use of the Informant object introduces an additional level of indirection towards obtaining the information for the action execution.

8 EMPIRICAL EVIDENCE

Pattern Mining Method

The anti-patterns and the pattern discussed in the paper came up during the project of a software engineering course that took place in the second semester of 2018-2019. The goal of the project was to develop a simple Latex editor. The students of the course formed 61 different development groups consisting of 2-3 people. Each group developed its own project. The overall duration of the case study was 10 weeks.

44 out of the 61 development groups used the Command pattern to structure the editor, with respect to commands that correspond to the required functionalities. The groups used the pattern similarly. However, the groups used different ways for configuring the Command objects with the appropriate data. Some of these solutions are inline with the observed anti-patterns, while some others conform with the observed pattern; specific percentages are reported later in the detailed discussion of the anti-patterns and the pattern.

Anti-Patterns and Pattern Uses

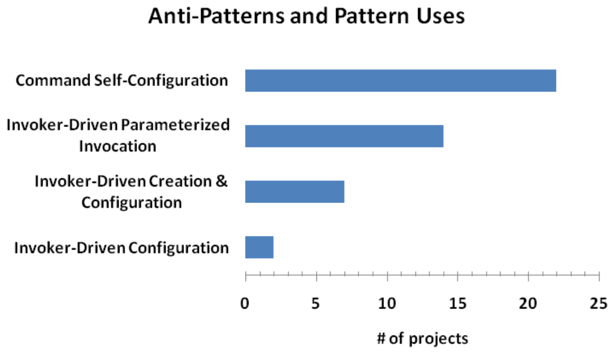


Figure 13: Anti-patterns and pattern uses in the software engineering course project.

Figure 13, summarizes the anti-patterns and pattern uses in the context of the software engineering project. Most of the groups used the Command Self Configuration pattern. In particular, 22 groups used the pattern. Overall, that is 36% of the groups that enrolled in the software engineering course and 50% of the groups that used the Command pattern in the context of the project. The most frequent anti-pattern is Invoker-Driven Parameterized Invocation. Specifically, 14 groups used this anti-pattern, i.e., 21% of the groups that enrolled in the software engineering course and 31% of the groups that used the Command pattern in the context of the project. The Invoker-Driven Creation and Configuration and the Invoker-Driven Configuration come next with 7 and 2 uses, respectively.

9 CONCLUSION

This paper revisited the GoF Command pattern, concentrating on the configuration of Command objects in the case where the required data become available as soon as the respective actions should be executed. The paper discussed frequent mistakes when dealing with this problem in the form of anti-patterns. As an alternative to the anti-patterns, the paper introduced a pattern that deals with the problem. The anti-patterns and the pattern came up during the project of a software engineering course. A interesting future research direction would be to look for anti-patterns and pattern uses in the context of open-source projects. Another interesting research direction is to investigate common pitfalls and mistakes in the use of other popular patterns.

ACKNOWLEDGMENTS

Many thanks to the shepherd of the paper, Mouna Abidi, and to the members of the writers workshop for their valuable comments.

REFERENCES

[1] Mouna Abidi, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2019. Anti-Patterns for Multi-Language Systems. In *Proceedings of the 24th ACM European Conference on Pattern Languages of Programs (EuroPLoP)*. 42:1–42:14.

[2] Apostolos Ampatzoglou and Alexander Chatzigeorgiou. 2007. Evaluation of Object-Oriented Design Patterns in Game Development. *Information and Software Technology* 49, 5 (2007), 445–454.

[3] Lerina Aversano, Gerardo Canfora, Luigi Cerulo, Concettina Del Grosso, and Massimiliano Di Penta. 2007. An Empirical Study on the Evolution of Design Patterns. In *Proceedings of the 6th Joint European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE)*. 385–394.

[4] James M. Bieman, Greg Straw, Huxia Wang, Willard P. Munger, and Roger T. Alexander. 2003. Design Patterns and Change Proneness: An Examination of Five Evolving Systems. In *Proceedings of the 9th IEEE International Symposium on Software Metrics (METRICS)*. IEEE Computer Society, Washington, DC, USA, 40–50.

[5] William Brown, Raphael Malveau, Hays McCormick, and Thomas Mowbray. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* Wiley.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1994. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley.

[7] Brian Huston. 2001. The Effects of Design Pattern Application on Metric Scores. *Journal of Systems and Software* 58, 3 (2001), 261–269.

[8] Salman Khwaja and Mohammad Alshayeb. 2016. Survey On Software Design-Pattern Specification Languages. *ACM Compututing Surveys* 49, 1 (2016), 21:1–21:35.

[9] Andrew Koenig. 1995. Patterns and Antipatterns. *Journal of Object Oriented Programming (JOOP)* 8, 1 (1995), 46–48.

[10] Ralf Laue. 2017. Anti-Patterns in End-User Documentation. In *Proceedings of the 22nd ACM European Conference on Pattern Languages of Programs (EuroPLoP).* ACM, 20:1–20:11.

[11] L. Prechelt, B. Unger, W.F. Tichy, P. Brossler, and L.G. Votta. 2001. A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions. *IEEE Transactions on Software Engineering* 27, 12 (2001), 1134–1144.

[12] Lutz Prechelt, Barbara Unger-Lamprecht, Michael Philippsen, and Walter F. Tichy. 2002. Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance. *IEEE Transactions on Software Engineering* 28, 6 (2002), 595–606.

[13] Marek Vokac. 2004. Defect Frequency and Design Patterns: An Empirical Study of Industrial Code. *IEEE Transactions on Software Engineering* 30, 12 (2004), 904–917.

[14] Peter Wendorff. 2001. Assessment of Design Patterns during Software Reengineering: Lessons Learned from a Large Commercial Project. In *Proceedings of the IEEE European Conference on Software Maintenance and Reengineering (CSMR).* 77–87.

[15] Apostolos V. Zarras, Georgios Mamalis, Aggelos Papamichail, Panagiotis Kollias, and Panos Vassiliadis. 2018. And the Tool Created a GUI That was Impure and Without Form: Anti-Patterns in Automatically Generated GUIs. In *Proceedings of the 23rd ACM European Conference on Pattern Languages of Programs (EuroPLoP).* 24:1–24:8.